ROB313 Assignment 3

Ethan Rajah
1006722370

I. Objectives

The primary objective of this assignment is to learn common implementations of gradient descent, which is used for neural networks and other significant machine learning frameworks. Gradient descent involves minimizing the loss of the model by taking the negative gradient of the loss, which provides the direction of steepest descent, and updating an estimate for the weights using set gradient. This process is repeated until the weights reach the minima of the loss function, thus being the optimal set of weights for modeling the problem. This specifically involves implementing full batch gradient descent, which involves using the entire dataset to determine model parameters and predicting on the test set with the resulting weights that minimize the loss. Furthermore, stochastic gradient descent (SGD) is introduced, where batches of data are used to determine the gradient at each iteration, rather than using the full dataset. This is used as it improves runtimes of training on large datasets and has been proved to be an unbiased approximation of the full gradient descent, meaning that it will still trend towards minima, especially with smaller learning rates. SGD with batch sizes of 1 and 10 are used in this assignment to compare the effects of increasing batch size and its trade-off on runtime.

A significant factor that affects the accuracy of a gradient descent model is the learning rate. The learning rate is equivalent to the time step used for each gradient descent iteration. There are advantages and disadvantages associated with learning rates that are too small or too large, which is explored further in this assignment. Also, the concept of momentum is implemented with stochastic gradient descent to show the convergence effects of using the prior gradient descent measurement within the current update to determine the weights. Within question 1, the gradient descent of the least squares loss function is explored, however, log likelihood gradient descent and a sigmoid activation function is used within question 2 to train on a classification set. The latter provides an introduction to logistic regression models, which are very common within the field.

II. Code Structure and Strategies

The PumaDyn dataset is used for the regression within question 1, whereas the iris dataset, specifically the classification of the iris virginica flower is used for question 2. As per the assignment instructions, question 1 only uses the first 1000 points of the training data to predict on the test set and question 2 uses a combination of the training and validation sets to do its prediction.

Question 1:

The structure for implementing full batch gradient descent and stochastic gradient descent was very similar. A range of learning rates were chosen to study the effects of training with rates that are too small or too large. Specifically, 6 learning rates in the range of 0.000001 to 1 are used to train and test on the data. As mentioned in the assignment, the optimal loss for this dataset was determined within assignment 1 using SVD. To compare the results of the gradient descent algorithm to the SVD results from assignment 1, the optimal loss for the training data was recalculated using SVD and plotted as a dashed line for each of the loss iteration plots generated within the algorithm. Moreover, an iteration count of 10000 was used to indicate the number of times the gradient descent update would run per learning rate. Within the main loop of the function, which iterates through the six learning rates presented, we compute 10000 iterations of gradient and stochastic descent. Since we are using least squares loss for this problem, the gradient function is written as the following:

$$\nabla L = \frac{1}{N} X^T (Xw - y)$$

where **X** is the N by (D+1) training data matrix, **w** is the (D+1) by 1 column vector of weights and **y** is the N by 1 column vector of labels in the feature space. For each iteration of full batch gradient descent, we compute the gradient using the current set of weights and the full dataset and then update the weights using the result. More specifically, the weight vector update is the difference of the previous weight vector and the learning rate multiplied by the gradient. This is simply a rearrangement of the formal definition of the derivative, as shown in lecture. With the updated weights, I compute the average least squares loss and store it within an array. This provided me with the ability to plot how the least squares loss changed as the iterations of gradient descent increased.

Since the study of convergence for different learning rates and gradient descent methods is of interest, I used the math.isclose function to print the time and epoch number when the loss of the training sequence converges to the optimal loss outlined by the SVD computation at the beginning of the function. Once the 10000 iterations are complete, the final set of weights for the given learning rate are used to calculate the test RMSE using the test data. With this, the least squares loss over the iterations is plotted for comparisons between varying learning rates. Once all of the learning rate test RMSE values are computed, I determine the minimum RMSE and distinguish the corresponding set of weights and learning rate as the model parameters.

The only difference between this implementation and the implementation of stochastic gradient descent is the portion of the dataset used to compute the gradient. To implement the feature of a user being able to set their desired batch size for SGD, I use the np.random.choice function to generate an array the size of the batch desired, from indices in the range of 0 to the number of elements of the dataset. This array is then applied to the training set to define the choice of data points being used to compute the gradient at that iteration. It is worth noting that for the computation of the convergence time, I needed to widen my convergence range from 1^{-3} to 1^{-2}. The reason for this will be emphasized within the results and discussion section.

To implement momentum into the SGD algorithm, the gradient update step needed to be revised. For each iteration, the gradient approximation using momentum as provided in the assignment handout is used for the update step. This computed gradient is stored to be used in the next iteration as the previous gradient. The previous gradient is scaled by the momentum hyperparameter to weigh the importance of the previous gradient as we make a step and recalculate the gradient of the data batch. This can be useful in helping the model to not branch off in a completely new direction from the previous gradient direction when doing the update. However, this a priori influence is dictated by the momentum hyperparameter. A range of momentum values on the range of 0.1 to 0.9 is iterated over the different choices of learning rates to determine the best combination of the two in predicting on the test set. The loss over iteration plots for each of these combinations are plotted for further analysis of model behavior. Aside from these additions, the structure of the code remains the same as the implementations for full batch and stochastic gradient descent.

Question 2:

The structure of the full batch and stochastic gradient descent implementations presented in question 1 were modified for question 2. Now, instead of a least squares loss analysis, the focus is shifted to a statistical training method using the maximum likelihood of a Bernoulli likelihood function and sigmoid activation function to determine the gradient to be applied to the weight update step. The key likelihood

formulations are presented within the assignment handout and the scipy.special.expit function was used as the sigmoid activation function to be applied to the regression to generate the logistic sigmoid. Once again, a set of learning rates in the range of 0.000001 to 0.01 were iterated upon to assess the effects of varying learning rates on the negative log likelihood. The gradient descent update process follows that of the full batch and SGD implementations but using the log likelihood function instead of the least squares loss to determine the gradient. Moreover, the negative log likelihood of each of the 10000 iterations is stored within an array and is plotted against the iteration number for each learning rate. Also, since the iris dataset is a classification set, the accuracy of the final weights after the 10000 iterations is computed, rather than the RMSE. Since an identification of False is 0 and an identification of True is 1, I calculate the accuracy by converting the logistic model test results into boolean values (True if the value is greater than 0.5, False otherwise) and comparing it to the expected test results. This comparison is then averaged over the number of data points in the test set. After this set of computations is complete for all of the learning rates specified, the weights and learning rate that maximize the model accuracy are distinguished upon and their respective test accuracy is returned. This process is repeated for SGD, where we define a batch size of 1 (as specified for the question) and use set batch to compute the gradient for each iteration. This is the exact same process as the SGD implementation within question 1. With both full batch and stochastic gradient descent complete, the negative log likelihood for the optimal models are plotted together against the number of iterations to compare performance.

III. Results and Discussion

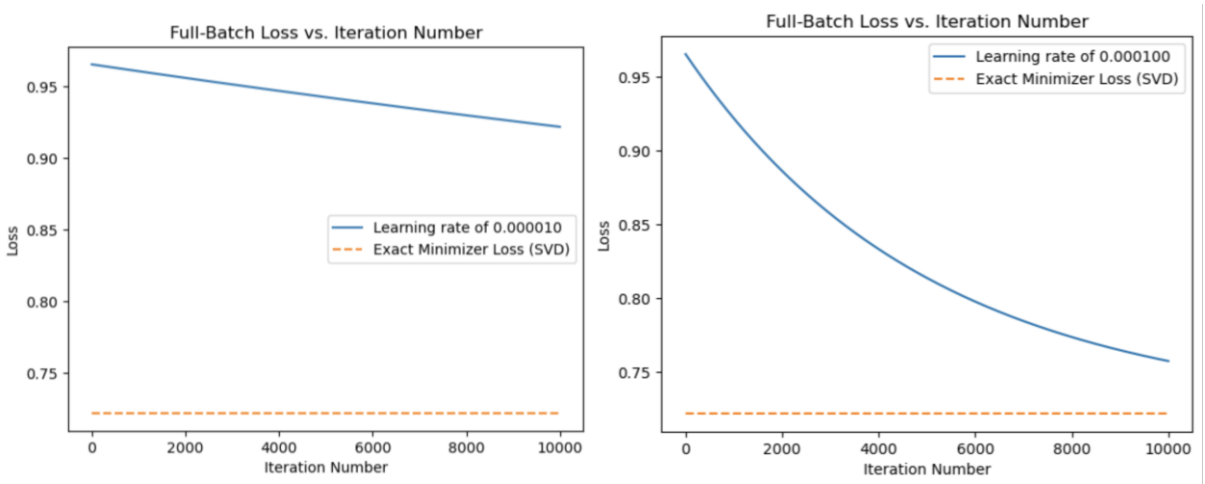

Figure 1: [cont. on next page]

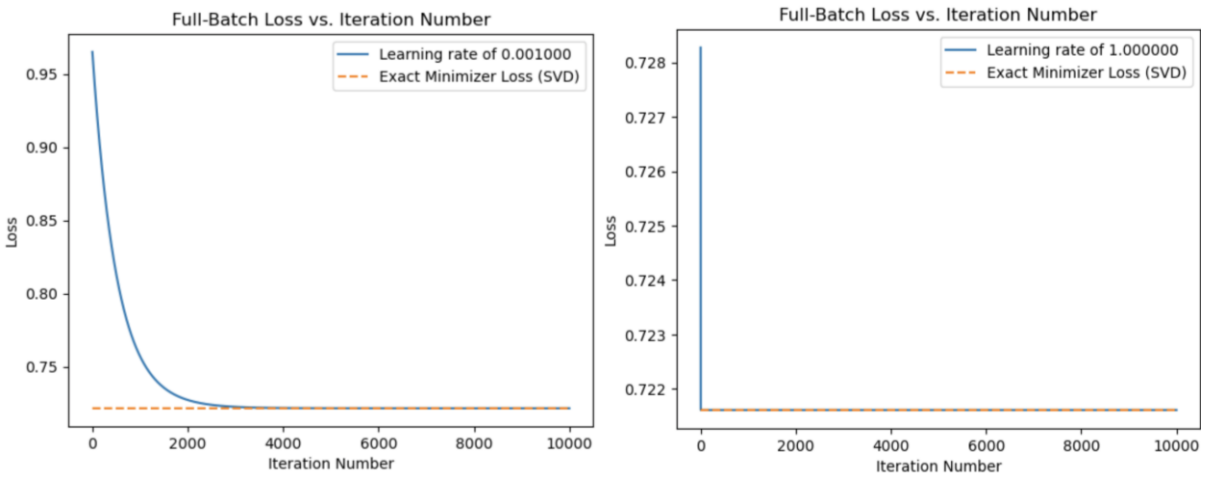

Figure 1: Full batch least squares loss vs. iteration number for varying learning rates

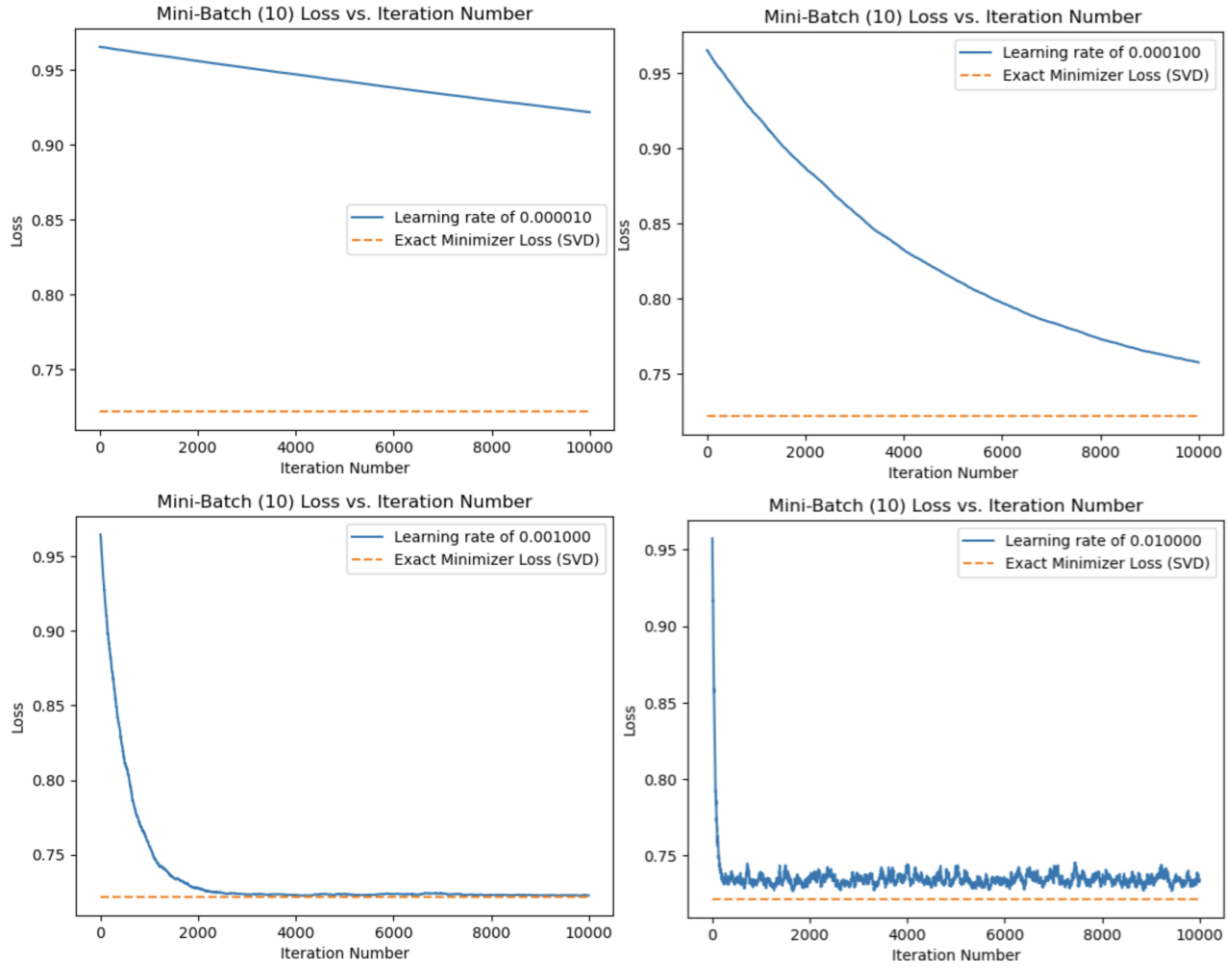

Figure 2: Size 10 mini batch least squares loss vs iteration number for varying learning rates

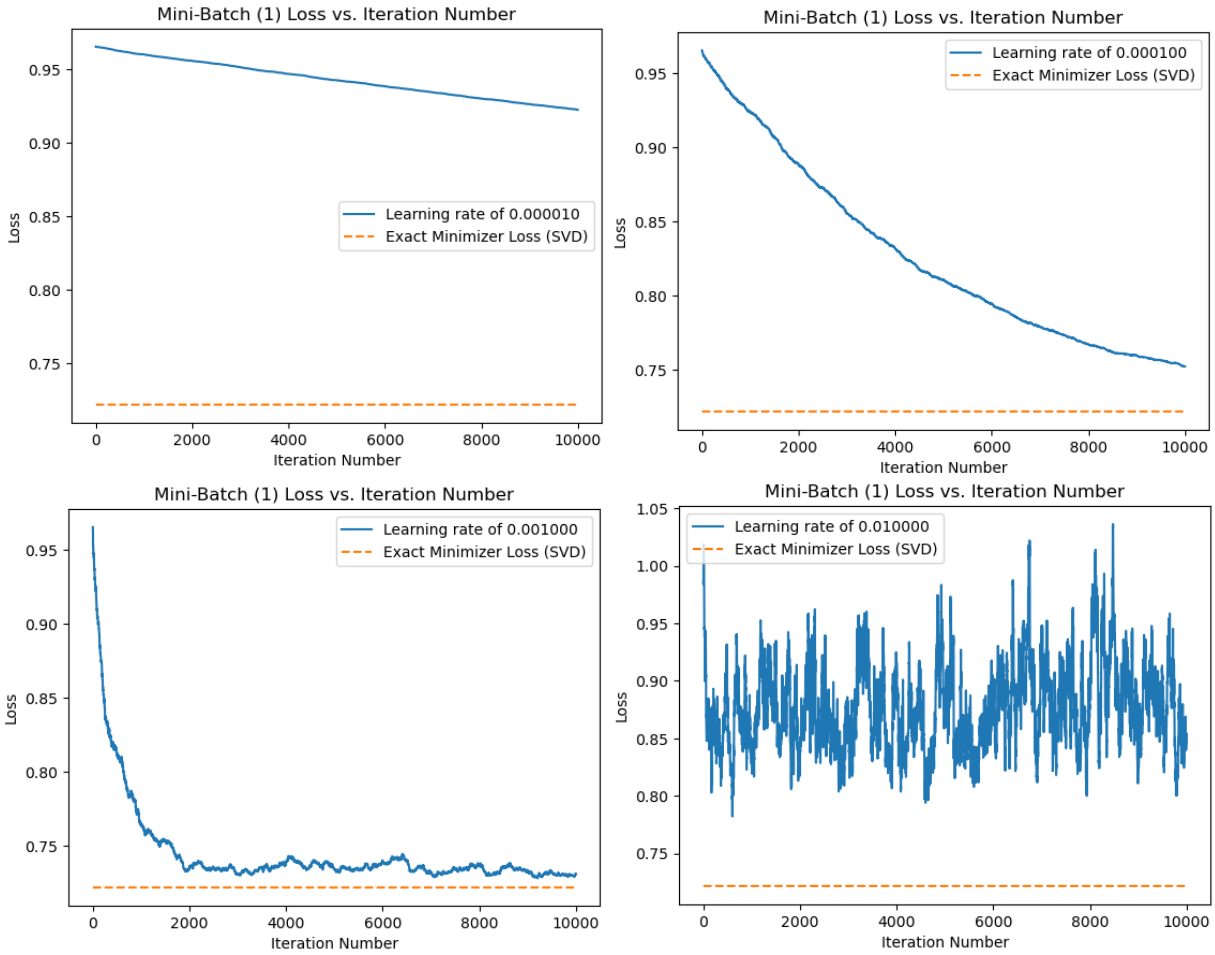

Figure 3: Size 1 mini batch least squares loss vs iteration number for varying learning rates

Time (s)	Epoch	Learning Rate	Batch Size	Momentum
0.259	3174	0.001	Full Batch	N/A
0.0159	316	0.01	Full Batch	N/A
0.0005	1	1	Full Batch	N/A
0.163	222	0.001	Mini Batch (10)	N/A
0.537	7059	0.001	Mini Batch (1)	0.1
0.317	4558	0.001	Mini Batch (1)	0.5
0.346	4347	0.001	Mini Batch (1)	0.65
0.323	4721	0.001	Mini Batch (1)	0.9

Table 1: Convergence and epoch trends for full batch and mini batch gradient descent with and without momentum.

	Test RMSE	Learning Rate	Momentum
Full Batch	0.871	0.01	N/A
Mini Batch (1)	0.869	0.001	N/A
Mini Batch (10)	0.872	0.001	N/A
Mini Batch (1) with Momentum	0.869	0.001	0.1

Table 2: Test RMSE results for full batch and mini batch GD/SGD using optimal hyperparameters.

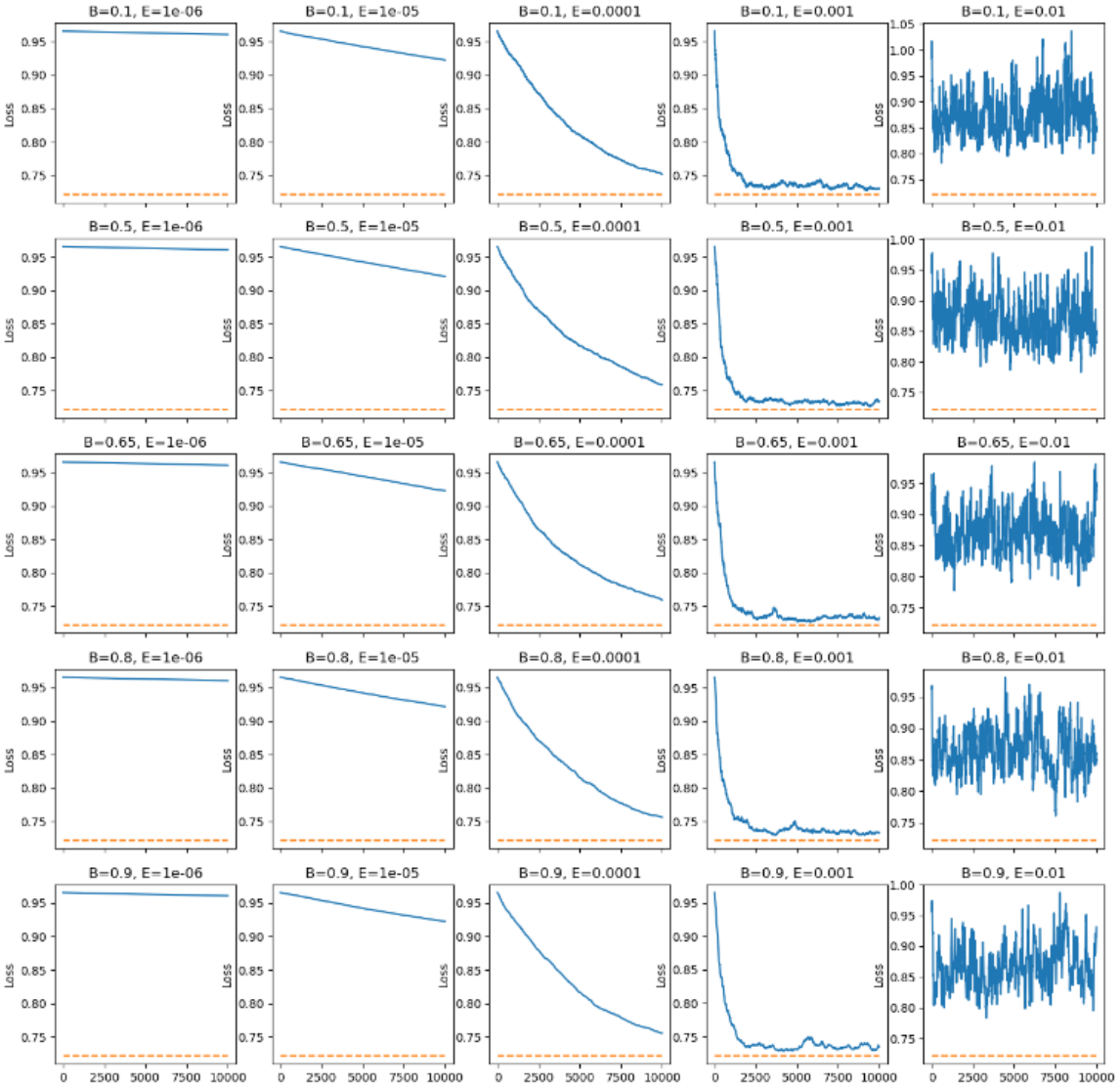

Figure 4: Size 1 mini batch least squares loss vs iteration number per momentum and learning rate choice

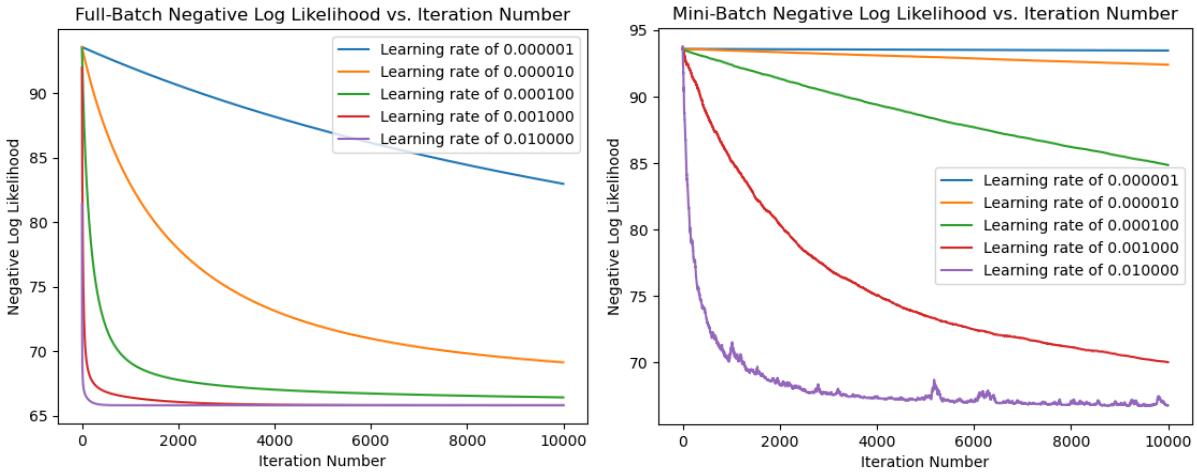

Figure 5: Full batch and mini batch (1) negative log likelihood vs. iteration number

	Test Accuracy	Negative Log Likelihood	Learning Rate
Full Batch	0.733	-6.913	0.01
Mini Batch (1)	0.733	-7.369	0.001

Table 3: Test accuracy and negative log likelihood results for full batch and mini batch GD/SGD using optimal hyperparameters.

Question 1:

Figures 1-3 show the gradient descent loss results with respect to the iteration number for both full batch and mini batch (of size 1 and 10) implementations. It can be seen within these plots that the full batch loss converges much more smoothly than what can be seen by mini batch, particularly for mini batch size 1. This follows the expected behavior of the algorithm because with the mini batch we are sampling much less points, thus, we expect to be less precise in the gradient descent update than that of the full batch implementation as we trend towards the minima. Within **Figure 1**, it is easy to notice that as we increase the learning rate, the decay of loss grows significantly. For very small learning rates in the full batch descent, we are unable to converge within the 10000 iteration limit, however, as we increase the learning rate to 1 or larger, the full batch loss convergence is instant. A similar result is seen in **Figure 2** and **Figure 3**, where very small learning rates result in no convergence, whereas as the learning rates increase, more noise can be seen with faster convergence. This is because these figures depict the results for SGD mini batch sizes of 1 and 10. With SGD, we tend to use smaller learning rates than what is optimal for full batch because we are sampling from the full dataset, so we are more likely to compute a gradient that does not point to the minima of the full dataset, resulting in the noise shown in the plots. This means that by using smaller learning rates, we can increment the gradient descent more slowly and thus more precisely to avoid this noise. In comparing the use of a learning rate of 0.01 for both mini batch size 1 and size 10 in **Figure 2** and **Figure 3**, one can notice the large difference in noise and therefore convergence results. Since we sample 1 point per iteration of SGD for mini batch size 1, using the learning rate of 0.01 results in extreme loss fluctuations on the training set as the number of iterations grows. As we use larger batch sizes, in this case a batch size of 10, we see that the noise is less severe, although still apparent. The presence of this noise was the reason for widening the convergence range by a factor of 10 for SGD. Once again, this is because with larger batches per iteration, we have more

data to compute the gradient with respect to, therefore resulting in a gradient direction that is more accurate to the full batch gradient direction. These three figures show that there is a range of learning rates that should be considered when developing a model based on gradient descent. If the learning rate is too small, it will take very long to converge to a minima, but if it is too large, it becomes easier for the gradient descent algorithm to lose track of the direction of steepest descent towards the minima, particularly for mini batch SGD. For full batch GD, too large of a learning rate will cause the loss to begin at optimal loss but then become extremely large (asymptotic to infinity). This was observed in my code, where this situation would result in overflow problems and an incomplete plot due to the asymptotic behavior to infinity.

Figures 1-3 show the convergence of the loss to the optimal loss found by using SVD. A more analytical description of the convergence trends for full batch and mini batch GD/SGD is shown in **Table 1**. We see that full batch gradient descent converges faster than mini batch SGD on average. This is expected as we've seen that as we use smaller batches, we need smaller learning rates which result in slower convergence rates to avoid the noise. However, with full batch, since we have the entire dataset to compute the gradient with respect to, we avoid the noise and thus larger learning rates, which cause faster convergence, can be used. The shortest convergence time is given for the full batch method with a learning rate of 1. This is expected as we saw in **Figure 1** that for larger learning rates for full batch, we converge almost instantely. **Table 1** also allows for a comparison between convergence and epoch number. The epoch number of mini batch size 10 with a learning rate of 0.001 is seen to be the smallest as the epoch number is N/10, where N is the number of iterations done until convergence. In the case of full batch and mini batch with size 1, the epoch number is N. From this, it is clear that using a mini batch SGD with size 10 or greater will significantly reduce the epoch number compared to using the other methods, which is the optimal choice for runtime behavior. By limiting the batch size being used to compute the gradient, we can improve runtime as shown in lecture when introduced to SGD, which will allow for faster response of other computations within the model program. Moreover, we've seen through **Figure 2** that using this approach has no effect on the steady state convergence of the model in comparison to the full batch method, as long as an appropriate learning rate is chosen. Overall, the epoch number results show that using a mini batch SGD approach can significantly reduce computation per cycle.

An interesting result was that for SGD with mini batch size 1, there was no convergence. However, when we include momentum into the computation, we are able to converge by reducing noise (false minima direction) when computing the gradient. The effects of using momentum with different learning rates is seen in **Figure 4**. It is difficult to see a difference between the choice of the momentum hyperparameter and a fixed learning rate from these plots. However, when comparing it to **Figure 3**, one can notice that momentum aids in reducing the noise, allowing for the convergence to be more steady. This behavior is expected as the momentum hyperparameter scales the weighting of the previous gradient term within the update step. This allows us to take into account the direction of minima presented in the previous iteration and average it with the direction presented in the current step to attempt to be more accurate in traveling in the direction of steepest descent. This is clearly shown in **Table 1**, where with no momentum there is no convergence data, but with momentum, the noise decreases allowing for the program to indicate that we did converge to the optimal loss result. Since the mini batch is of size 1, the epoch number remains high compared to the other methods. It is also worth mentioning that the SGD with momentum only converges for a learning rate of 0.001, which can be seen in the plots of **Figure 4**. Moreover, since we use a mini batch size of 1 with a small learning rate and added computational cost from the momentum term in the gradient, the convergence time is among the largest within the data presented in **Table 1**.

The test RMSE results from using the optimal hyperparameters computed for each method is presented in **Table 2**. All of the test RMSE results were very close to one another; I expected that the full

batch RMSE would provide the smallest RMSE value, however, mini batch with momentum did. Since these values are very close to one another, it is likely that the small noise from the optimal loss of the training set ended up slightly improving the accuracy when the mini batch model was applied to the test set as the noise acts as a regularization to reduce generalization error. Moreover, the random batch selection of points from the training set to be used for the gradient descent may have resulted in small changes in the weight vector in comparison to the full batch weights that caused the test RMSE to be slightly better. Also, the larger batches of data may be resulting in overfitting of the weights, which increases the error of the model when applied to the test set. Comparing the three variants of SGD within **Table 2**, it can be seen that a learning rate of 0.001 results in optimal performance for each version. This was expected due to the minimal presence of gradient noise in the loss iteration plots in **Figures 2-3**. Furthermore, a momentum hyperparameter of 0.1 was found to provide the optimal test RMSE loss. Of the three variants, the mini batch of size 1 with momentum performed the best on the test set, but only by a very small amount. Although my intuition initially suggested that a larger batch size would perform the best on the test set, the presence of overfitting the weights based on the larger use of the training set and the noise in the mini batch are most likely to be what caused the slightly better performance of size 1 mini batch. With this, we saw that the use of momentum allows us to be more precise with the convergence of the loss in the training set when performing SGD (reduce noise). Based on this, it was expected that momentum would improve the model accuracy, which turned out to be true, even if it was by a very small amount. To note, the RMSE results are presented with greater decimal accuracy when running the code. For future testing, it would be worth iterating through more SGD variants, particularly trying mini batch with larger batch sizes and using momentum to become more aware of the use cases for each version when training and testing on various data.

Question 2:

The full batch and size 1 mini batch negative log likelihood against iteration number plots are presented in **Figure 5**. As with question 1, many learning rates are considered and are plotted together to show their corresponding convergence rate. Once again, there is a clear range of learning rates that were determined to be appropriate for training the model. **Figure 5** shows this as some of the learning rates were too slow to converge within the 10000 iteration constraint. Another similarity to question 1 was that as we increase the learning rate, the mini batch loss decay begins to display more and more noise, which we've seen is a result of the gradient direction alternating from the true minima direction when applying the mini batch update. It was also interesting to notice that when comparing the full batch and mini batch plots, the smaller learning rates for mini batch decay much slower than what is seen for full batch. This was not observed in **Figures 1-3**, which indicates it may be a result of using a negative log likelihood representation rather than least squares loss. It is likely that the use of batches do not provide a large enough set of i.i.d observations for the log likelihood to change by a significant amount with very small learning rates.

Since the model was trained on the classification dataset iris, the test accuracy and negative log likelihood for the test data is presented in **Table 3**. It was found that for both methods, the test accuracy was the same. This is a good sign as it shows that SGD, while saving on runtime, is also able to provide reliable results for the model. As expected from the discussions presented in question 1, the optimal learning rate for the full batch method was greater than that of the mini batch method, specifically by a factor of 10. The test negative log likelihood of the mini batch method was determined to be about 0.4 less than that of the full batch method. This provides us with more information on the performance of the model, which is why it is a more preferable performance metric for classification. These results are more precise than comparing test accuracies, especially when the accuracies are very similar or exact for a particular set of test data. This metric allows us to assess performance qualities that are not observable directly through the test accuracy

alone. For log likelihood loss representation, it is giving the log likelihood that the true output observations y, are observed by the model parameters given, w and X. Thus, the lower the negative log likelihood, the better the model is at predicting on test data. This means that size 1 mini batch is more preferable for this dataset as it has a smaller test log likelihood, which is information that we would not have been able to identify if we solely used the accuracy metric. Overall, this provides insight on the applicable nature of log likelihood to classification problems as it allows one to narrow in on more precise accuracy details of using different methods to train the model.

For additional information when completing the assignment, I plotted the training negative log likelihood loss results for the optimal parameters presented in **Table 3** within **Appendix A: Figure 6**. This allows for an easier visualization of the convergence speed between the full batch and mini batch methods when training the model. Although the full batch method converges much faster, we see through the test negative log likelihood that the mini batch method becomes more precise for classification on new data.

Appendix A:

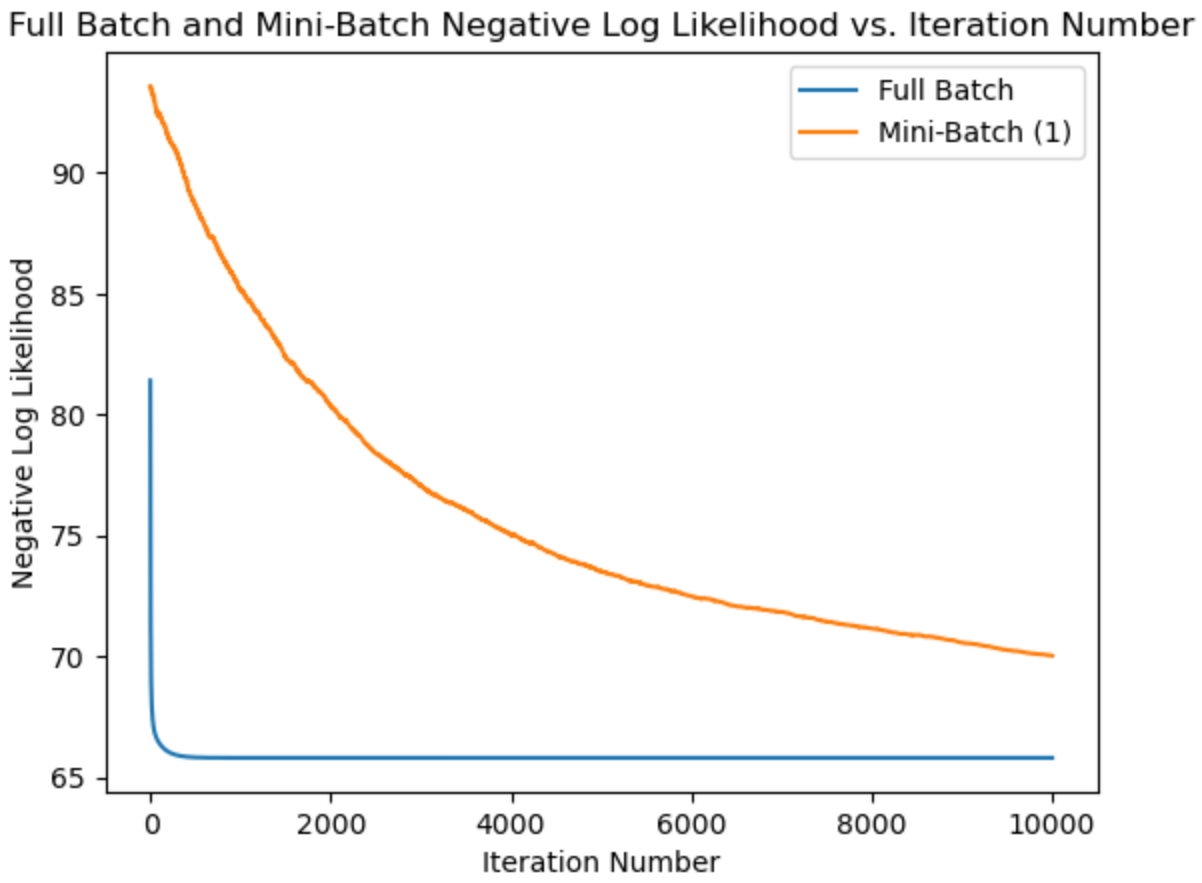

Figure 6: Negative log likelihood comparison of full batch and mini batch (1) gradient descent performance. The optimal weights that maximized accuracy were used for plotting.